BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

GENERATION OF BINARY PREFIX TREE ACCORDING TO PROBABILISTIC PARAMETERS

GENEROVÁNÍ BINÁRNÍHO PREFIXOVÉHO STROMU PODLE PRAVDĚPODOBNOSTNÍCH PARAMETRŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ ŽENČÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ MATOUŠEK,**
VEDOUCÍ PRÁCE

BRNO 2017

Zadání bakalářské práce/19663/2016/xzenca00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Ženčák Tomáš**
Obor: Informační technologie
Téma: **Generování binárního prefixového stromu podle pravděpodobnostních parametrů**
Generation of Binary Prefix Tree According to Probabilistic Parameters
Kategorie: Algoritmy a datové struktury

Pokyny:

1. Seznamte se s datovou strukturou trie (binární prefixový strom) a pravděpodobnostními parametry použitými pro její reprezentaci.
2. Nastudujte způsob generování trie podle pravděpodobnostních parametrů.
3. Navrhněte alternativní způsob generování trie podle pravděpodobnostních parametrů.
4. Navržený alternativní způsob generování trie implementujte.
5. Experimentálně ověřte kvalitu implementovaného způsobu generování trie a porovnejte ji s dostupnou implementací.
6. Diskutujte dosažené výsledky a možnosti dalšího vylepšení navrženého řešení.

Literatura:

- D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," in IEEE/ACM Transactions on Networking, vol. 15, no. 3, pp. 499-511, June 2007.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Matoušek Jiří, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstract

The goal of this work is to create a prefix set generator which will be capable of generating a prefix set from parameters specified in the ClassBench toolset. This work describes a possible approach to generation, as well as the final algorithm for prefix set generation. The resulting generator is able to generate prefix sets whose average deviation from the target parameters is typically orders of magnitude lower than the deviation of sets generated by ClassBench.

Abstrakt

Cílem této práce je vytvořit generátor prefixových sad, který bude schopný vygenerovat prefixovou sadu na základě parametrů specifikovaných v sadě nástrojů ClassBench. V této práci je popsaný možný přístup ke generování, jakožto i konečný algoritmus generování prefixové sady. Vytvořené řešení umožňuje generovat sady prefixů, jejichž průměrná odchylka od požadovaných parametrů je typicky o několik řádů nižší než odchylka sad generovaných nástrojem ClassBench.

Keywords

sady IP prefixů, generování, prefixový strom, ClassBench

Klíčová slova

IP prefix sets, generation, trie, ClassBench

Reference

ŽENČÁK, Tomáš. *Generation of Binary Prefix Tree According to Probabilistic Parameters.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Matoušek Jiří.

Generation of Binary Prefix Tree According to Probabilistic Parameters

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Jiří Matoušek from the Faculty of Information Technology, Brno University of Technology. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tomáš Ženčák
May 16, 2017

Acknowledgements

I would like to thank Ing. Jiří Matoušek for his patience and time, invaluable input and advice on writing a thesis, and supplying the scripts which I tweaked to automate the creation of the charts showing the error of the resulting prefix sets.

Contents

Chapter 1

Introduction

In the last several years, the number of devices connected to the Internet has been growing rapidly, as has demand for ever higher link speeds. This means that backbone router routing tables have been growing, but at the same time, lookup has to be quicker.

When a packet arrives to a router, the router must decide where to send it. this decision is done according to the routing table, which consists of a set of prefixes, each with a corresponding interface to which a packet with destination address matching the prefix should be sent. However, there may be several prefixes of different lengths matching an IP address. For example, both 01* and 011* match 01110. In such cases, the router should select the most specific (i.e., the longest) prefix matching the address. This task is called longest prefix matching.

Therefore, it is necessary to develop new, faster algorithms for finding the longest prefix match. To judge improvements over existing algorithms, the new algorithms must be benchmarked. However, the performance of such lookup algorithms is dependent on characteristics of the IP prefix set used to benchmark them.

To get usable results from benchmarking, it is therefore necessary to test new developments on real IP prefix sets, or at least sets that have characteristics similar to real ones. Unfortunately, real IP prefix sets of sufficient size are contained in core routers, firewalls, and access control lists, and therefore they are often sensitive information.

IPv4 prefix sets from core routers are available [1][3], however, due to exhaustion of the IPv4 address space [4], the growth of IPv4 routing tables will stop eventually. But with the growing IPv6 adoption rate as well as the advent of Internet of Things and the associated increase in the number of devices connected to the Internet, the size of IPv6 routing tables will likely continue to grow. Due to the much larger size of IPv6 address space, IPv6 routing tables will encounter no such limit and are likely to reach and eventually far surpass the current sizes of IPv4 routing tables.

As routing table lookup in core routers requires very high performance, it is implemented in hardware [7]. This means that in case the performance becomes insufficient, it would be necessary to buy new hardware, which is a significant investment, and the performance of hardware that is bought today must be sufficient for routing table sizes anticipated in several years.

Given that tables of such size do not yet exist, the only way to create benchmarks that can give usable results is to use synthetic IP prefix sets. However, in order to be usable, these synthetic sets must have characteristics similar to real sets, and simple random generation will therefore not do. For this reason, it is necessary to find how to characterize IP prefix sets in such a way that lookup algorithms have similar performance on sets that are similar

in those characteristics, as well as how to generate synthetic filter sets which are as close as possible in these characteristics to the input characteristics (which were presumably extracted from a real prefix set.

The structure of this thesis is as follows: Chapter 2 describes the data structure that is the basis for representing prefix sets in the context of this work. Chapter 3 describes the existing toolset, one part of which is a filter set generator that uses trie-based parameters to represent the relationships of the prefixes in filter sets. Chapter 4 describes a new generation algorithm which uses the same parameters as inputs. Chapter 5 describes the final implementation of a trie-based prefix set generator. Chapter 6 shows the results acheived by the algorithms described in the other chapters. Chapter 7 discusses possible improvements that could be made to the resulting generator.

Chapter 2

Trie

A trie, also known as prefix tree, is a data structure mainly used for efficient searching in a set of keys that can be expressed as a string. This is often used for seraching and matching of strings, such as text prediction or spell checking.

However, the strings can contain arbitrary elements, and therefore any binary data can be stored inside a bitwise trie [2]. An example of such binary data are IP addresses. Bitwise tries are especially well suited for storing sets of IP addresses, as they tend to be organized hierarchically and tries work well for value sets with common prefixes.

A bitwise trie is a binary tree in which each node can have up to two children, with one representing a 0 and the other representing a 1. This means that a node of depth k represents a bit string of length k.

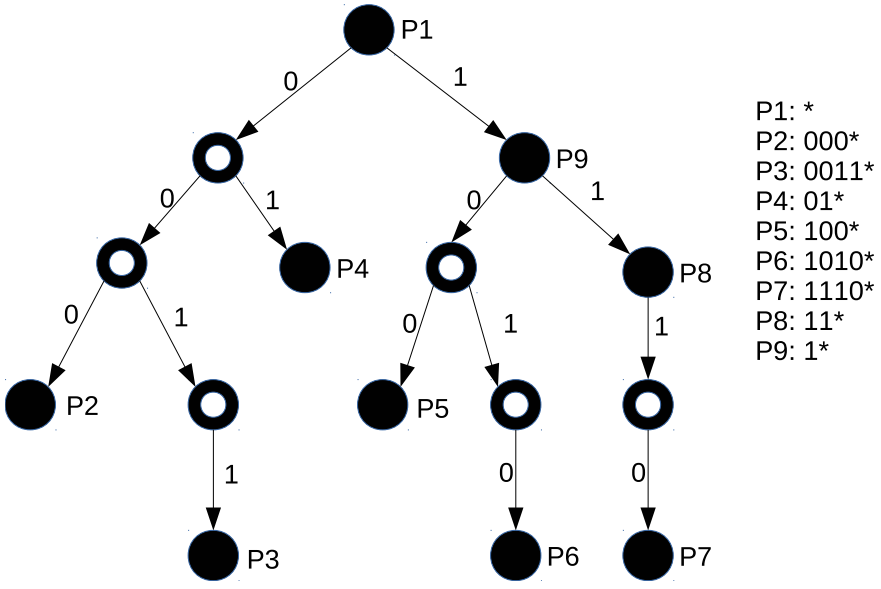

Figure 2.1: An example of a bitwise trie

Figure 2.1 shows an example of a bitwise trie storing nine prefixes. The trie consists of two kinds of nodes: prefix nodes, which represent an IP prefix, and structural nodes, which only serve to specifiy the structure of the trie. All leaf nodes are prefix nodes. To find the prefix a node represents, one simply has to walk the path from the root node to the given node and record which edges the path takes.

For example, the path to the node containing prefix P5 first takes the 1 edge, then the 0 edge, and finaly the 0 edge. Prefix P2 is therefore 100*. The path from the root node to itself is empty, and so is the prefix represented by it.

Searching for longest prefix match in a trie is fairly straightforward. One just has to iterate over the bits of the IP address to match, taking the appropriate branch at every node. If the current node does not have the corresponding branch, it means that the trie does not contain any prefixes matching the IP address longer than current depth. In that case, the last prefix node on the traversed path repreresents the longest prefix matching the given IP address.

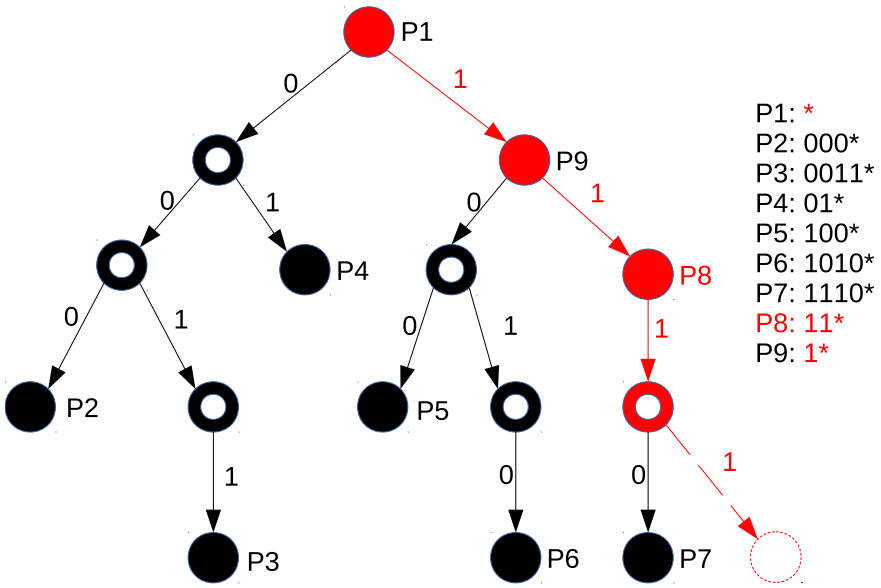

Figure 2.2: An example of IP lookup in a bitwise trie

Figure 2.2 shows an example of searching for the longest prefix matching the bitstring 111101:
Start at the root node, which contains the prefix P1. The first bit of the searched bitstring is 1, so take the 1 branch. This node contains the prefix P9. Next branch is also 1 and current node contains P8. Taking the next 1 branch, the current node has no associated prefix. The next bit of the bitstring is also 1, but the node has no 1 branch, and so there are no longer prefixes matching this bitstring. The longest matching prefix is therefore the last prefix encountered during the traversal, which is P8.

As can be seen from the above descriptions, the number of iterations necessary to find the longest matching prefix is at most the height of the trie. For values whose maximum length is fixed, such as IP prefixes, the upper bound on trie lookup is constant regardless of prefix count.

Chapter 3

Trie-based IP prefix generation

Taylor and Turner have specified a method of extracting relevant parameters from a filter set, which can then be distributed freely and can be used to generate a filter set with characteristics similar to the original, but contain no information about the specific filters in the original set. Thay also proposed a way to generate synthetic filter sets from the extracted parameters. Such a set would not expose any confidential information, but it would retain the characteristics important for testing packet classification implementations. They implemented these algorithms in a set of open source tools called ClassBench. [8]

As this thesis focuses on generating IP prefixes and not full filters, this chapter will only describe the generation of IP prefixes and omit information that does not relate to generation of IP prefixes.

The method of characterizing a filter set in ClassBench is based on creating a trie representing the IP prefixes in a filter set and then extracting characteristics describing the trie. Due to the fact that the trie is just a different representation of the prefix set, the characteristics of the trie also describe the prefix set.

3.1 Parameters

The relevant characteristics of the trie utilized in ClassBench are as follows:

Prefix nesting The number of unique prefixes matching a given IP address.

Skew distribution For a trie node with two children, skew is defined in Equation 3.1.

$$skew = 1 - \frac{Lweight}{Hweight} \tag{3.1}$$

where $Lweight$ is the weight of the lighter subtree and $Hweight$ is the weight of the heavier subtree. In case both subtrees have the same weight, which weight is which does not matter and the skew is equal to 0.
For nodes with less than two children, skew is undefined. The skew average is recorded for each level of the trie, and the result is the skew distribution.

Branching probability distribution Branching probability is the probability that the trie will branch at a given level. Nodes which have no children are excluded from the calculation. Branching probability is recorded for each level of the trie, resulting in te branching probability distribution.

Prefix length distribution This distribution records the number of prefixes of given length.

3.2 Parameter examples

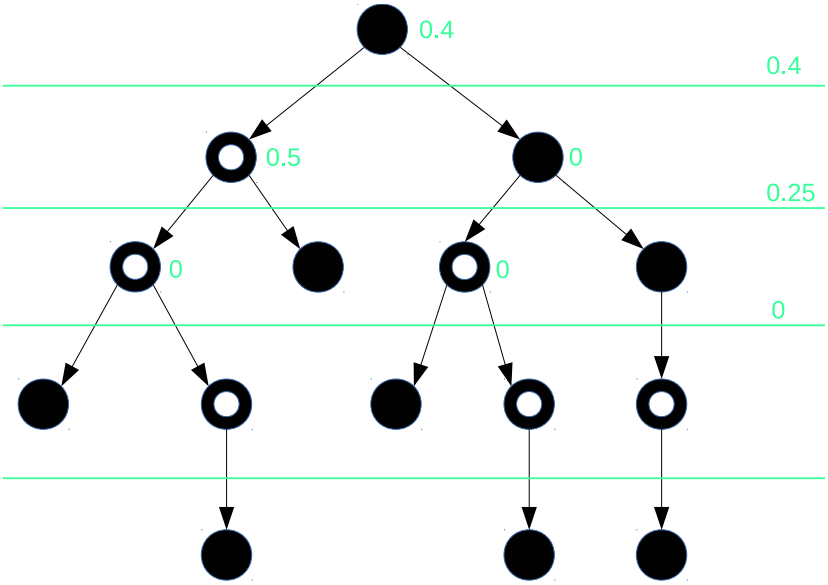

Figure 3.1: A trie with skews for single nodes and levels

In Figure 3.1, an example of calculating the skew distribution for the trie first used in Figure 2.1 can be seen. The root node has a skew of 0.4, as can be seen after evaluating Equation 3.1. The left subtree contains 3 prefixes and the right subree contains 5 prefixes. We can therefore substitute these values into the equation, yielding the following expression:

$$Skew = 1 - \frac{3}{5} = 1 - 0.6 = 0.4$$

As the root node is the only node in its level, the skew for level 0 is also 0.4.

On the next level, there are two nodes. Using the same calculation, we can see that the node representing the prefix 0* has a skew of 0.5, while the node representing the prefix 1* has a skew of 0. Note that while the 1 subtree contains more nodes than the 0 subtree, the skew distribution is defined as an ordinary arithmetic mean as opposed to a weighted arithmetic mean, and the skew for this level is therefore 0.25.

On level 2, one can note that only the nodes that have two children have skew calculated. Both the childless node and the node with just one child do not have defined skew, and they do not influence the skew of their level in any way.

On the subsequent levels, since there are no nodes with two children, the skew for those levels is also undefined.

Figure 3.2 illustrates the calculation of branching probability. Taylor and Turner specify both the probability that a node at a given level will have two children and the probability that it will only have one child in their branching probability distribution. However, seeing as leaf nodes are exclueded from the calculation, the probability that a non-leaf node will have one child is the complement of the probability that it will have two children and vice

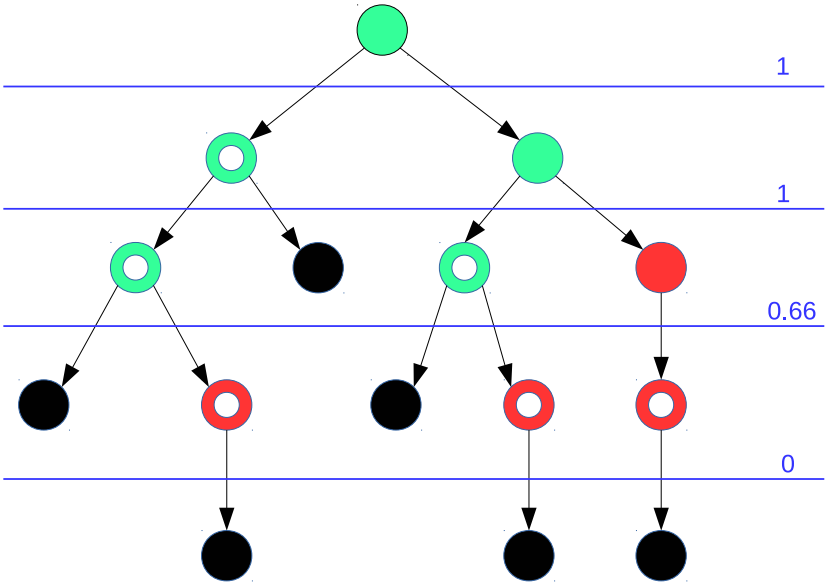

Figure 3.2: A trie with branching probabilties for given levels

versa. For the purpose of demonstrating branching probability calculation, I will therefore only specify the probability that a node on the given level will branch, i.e., will have two children. The probability that the node will have one child can be calculated as the shown probability subtracted from 1, and for the sake of simplicity, has been omitted from the figure.

In Figure 3.2, nodes which have two children are colored green, nodes with one child are colored red and nodes without children are left black. On levels 0 and 1, all nodes branch and the branching probability for both levels is therefore 1. On level 2, there are two nodes with two children, one node with one child and one leaf node. The leaf node does not influence the calculation in any way. The branching probability is then calculated as

$$\frac{branchingnodes}{non-leafnodes} = \frac{2}{3} = 0.67$$

. The probability that a node will have one child is $1 - 0.67 = 0.33$.

Level 3 contains 3 nodes with one child an two leaf nodes, but seeing as it contains no nodes with two children, the branching probalility for level 3 is 0. Level 4 only contains leaf nodes, and branching probability is therefore undefined.

3.3 Parameter representation

Taylor and Turner have also created a tool (Filter Set Analyzer) for extracting said characteristics from filter sets. However, as it is meant to represent filter sets as opposed to simple IP prefix sets, it also extracts other characteristics which are irrelevant for IP prefix set generation and therefore will mostly be ignored in this work.

The only characteristic relevant here will be Port pair class (PPC). It specifies the pair of source and destination port ranges, each split into 5 classes for a total of 25 Port Pair Classes. Note that this is only relevant for compatibility with ClassBench, otherwise it is of no use at all.

The Filter Set Analyzer analyzes a filter set and outputs a parameter file. These files contain the following values:

scale The number of filters in the original filter set.

protocols A transport protocol number, the probability that a filter targets this protocol and probabilities that a filter targetting this protocol falls into a given PPC.

prefix length probabilites For each PPC, a list of total prefix lengths (i.e., sum of source and destination prefix length), the probabilities that a filter matching this PPC has a given prefix length, and then a list containing source prefix lengths and the probabilities that a filter with the given total prefix length has source prefix of the given source length.

prefix nesting threshold The maximum source and destination IP prefix nestings in the original filter set.

skew The average skew in each level of source and destination tries. The computation ignores nodes where skew is undefined, and uses an unweighted arithmetic mean.

branching probability Branching probability is represented by two numbers: the probability that a node on a given level will have two children and the probability that it will have one child. Given that nodes that have no children are excluded from calculation, the sum of these two numbers is always 1. Due to this fact, one can always calculate one of the numbers from the other, and using both is redundant. In this thesis, branching probability will therefore always mean the probability that a non-leaf node has two children.

3.4 Generation

First, ClassBench generates *size* filters. It assigns a length to each IP prefix in the filter set based on the prefix length distribution from the parameter file. This defines the number of prefixes at each level of the resulting trie. Then it traverses the trie level by level and assigns nodes to filters whose IP prefix length matches the current depth. Afterwards, it distributes the filters with higher prefix lengths to child nodes, taking into account the desired skew and branching probability of the given trie level.

If the maximum prefix nesting value is reached, ClassBench simply forces all prefixes in the given subtree to end in its root, ignoring all other parameters.

Redundant filters in the resulting sets are discarded.

This approach has several problems. The above mentioned approach to limiting prefix nesting can introduce significant errors into the resulting set, and discarding redundant filters also affects the characteristics of the resulting trie.

The purpose of this thesis is to devise a better way, but due to the scope of this work, it shall be limited to generating a set of IP address prefixes from the probabilistic parameters extracted from a trie representing a real set of IP address prefixes as opposed to generating a set of 5-tuples representing complete filters.

Chapter 4

Proposed way

My proposal is similar to ClassBench's approach in that it processes the trie level by level. It starts with the trie containing the root node and sets the current level to 0 containing this single node.

It then processes each level in several phases, where each phase attempts to satisfy a different distribution. First, it distributes child nodes, upholding the branching probability distribution. It then ends prefixes adhering to the prefix length distribution. Next, it distributes prefixes to child nodes, attempting to get as close as possible to the skew specified for the given level.

Fter that, it proceeds to the next level. Seeing as this approach would not produce satisfactory results in cases when it would be necessary to decide according to the parameters of subsequent levels, the generator also keeps track of constraints. If it ever reaches a state in which it cannot get close enough to the desired parameters, it sets a constraint accordingly and then rolls back the process of generation to an earlier phase or even a previous level.

To get results that mimic the desired parameters as closely as possible, the generator does not simply generate the trie randomly, with bias according to the input distributions. It instead keeps track of characteristics in nodes that were already processed in the current level and the number of nodes left to process and it recalculates the probabilities before processing each node. This means that if possible, the input parameters are adhered to quite rigidly.

Each phase will be described in more detail below. In this description, the following terms will be used:

own prefixes A node's own prefixes are the prefixes contained directly in the node. For example, if the trie's root node has three own prefixes, it means that the prefix 0.0.0.0/0 is present in the trie three times.

weight Weight is the sum of the node's own prefix count and the weights of its children, if there are any.

available prefixes The available prefixes are the prefixes that can still be freely assigned as necessary. Each node starts with its available prefix count set to be equal to its weight and the available prefixes are then either assigned to be the node's own prefixes or distributed to the node's children. When the node is fully processed, its available prefix count must be zero. This means that the only option for a node with no children is for all available prefixes (and therefore it's whole weight) to become the node's own prefixes.

child nodes The nodes belonging to the next level.

one-child nodes Nodes that have one child.

two-children nodes Nodes that have two children.

parent nodes Nodes that have at least one child.

4.1 Child node distribution

This phase starts with a state in which all nodes of a given level were already created and had a number of prefixes assigned to them, which is their weight and, at this point, also their available prefix count. At the end of this phase, nodes for the next trie level are created and assigned as children to the current level's nodes so that the branching probability distribution is satisfied and other phases can successfully satisfy their constraints.

The distribution this phase is mainly concerned with is the branching probability distribution. First, the generator must find out how many nodes at this level will have two children and how many will have one child. Unfortunately, the branching probability distribution only specifies the percentages of one-child nodes and two-children nodes at each level. The generator cannot therefore calculate these numbers by simply multiplying the amount of nodes at the given level by the percentage from the branching probability distribution, as that would ignore the number of childless nodes, which is not recorded in the branching probability distribution. A different method must therefore be used.

The used method relies on the fact that the branching probability is calculated as follows:

$$branching\ probablity = \frac{two\text{-}children\ node\ count}{parent\ node\ count} \tag{4.1}$$

We only know the branching probability and we are interested in the two numbers used to calculate it. Both are natural numbers, so it is simply a matter of finding a fraction that is equal to the branching probability. We know that the parent node count cannot be greater than the count of nodes at this level. This makes it possible to search for the fraction by searching a Stern-Brocot tree, which is an infinite binary search tree containing all fractions between 1 and 0 [6]. Each node in the tree represents a fraction whose denominator is always higher than that of the fraction represented by its parent. This allows the generator to simply search the tree until encountering a fraction whose denominator is higher than the maximum possible count of parent nodes at the current level or finding a fraction that is exactly equal to the branching probability. The maximum possible number of parent nodes is limited by two factors:

1. the number of nodes at the current level
2. the number of these nodes that contain multiple prefixes

The first is obvious. The second stems from the fact that no node in the trie can have zero weight and nodes that have only one available prefix therefore cannot have two children, as a node must pass at least one prefix to each child. This limits the number of two-children nodes in the level and therefore also the maximum number of parent nodes, which should maintain a fixed ratio to the number of two-children nodes. The maximum number of parent nodes can therefore be calculated using the following formula:

$$min\left(\frac{1}{two\ child\ probability} * multiprefix\ node\ count, current\ level\ node\ count\right) \tag{4.2}$$

The found fraction's numerator represents the two-children node count and the denominator represents the parent node count. The two-children node count is therefore equal to the numerator of the found fraction, while one-child node count is calculated as $denominator - numerator$.

However, these numbers are not final. Since the Stern-Brocot tree only contains fractions in their most reduced forms, if the original branching probability was calculated from a reducible fraction, this way would overestimate the number of childless prefixes and cause trouble. The generator therefore assumes that the number of parent nodes is the greatest it can be while still maintaining the ratio of two-children nodes to one-child nodes. It sets a multiplier such that

$$multiplier = floor\left(\frac{max\ parent\ nodes}{found\ fraction\ denominator}\right) \tag{4.3}$$

If there is a constraint set for this level, it lowers the multiplier so that the constraint can be satisfied. The one-child node count and two-children node count is then multiplied by the resulting multiplier.

Next, the nodes are randomly split into sets of childless, one-child and two-children nodes so that each set has the size calculated previously. This splitting is, however, not final. There may still be adjustments necessary in order for the next phases to be able to satisfy their respective distributions.

First, the generator makes sure that it is not forced to end more prefixes than specified in the prefix length distribution. Due to the fact that childless nodes cannot pass their prefixes to the next levels, all the prefixes contained in childless nodes must be ended at the current level. The generator therefore checks how many prefixes would be forced to end with the current allocation and if this number is higher than the number of prefixes that should end at this level according to the prefix length distribution, the generator swaps nodes between sets so that it moves high prefix nodes out of the childless node set and low prefix nodes into it until the count of forcibly ended prefixes is lower than the count of prefixes that should end. In case it is unable to reach this state, it sets a constraint and rolls the generation back to the start of the previous level.

Next, it checks if there is a constraint indicating that there must be at least a certain number of multiprefix children. Not having enough multiprefix nodes can cause the generator to assume that the maximum possible number of parent nodes is too low, leading to a situation in which far too many nodes are childless or to a state in which there are childless nodes at a level at which no prefixes are supposed to end. If there is such a constraint, the generator transfers higher prefix nodes from sets with lower child count to ones with higher child count until it is possible to have at least the number of multiprefix children specified by the constraint.

After that, the generator creates the child nodes and assigns them to parents. Each child is at first assigned only a single prefix, unless a constraint specifies that there is a mandated number of multiprefix children, in which case corresponding number of children is given two prefixes.

Finally, the generator attempts to ensure that the target skew is reachable. If there is no constraint on the number of available prefixes in two-children nodes, the generator simply swaps nodes between two-children nodes and other node sets so that the number of prefixes in two-children nodes increases. While doing this, it checks the minimum and maximum reachable skew and if the desired skew is within those bounds, it stops. Otherwise, the generator finds the average of the minimum and maximum number of two-children node

prefixes and attempts to get the number of prefixes in two-children nodes to be as close to it as possible. This is done after the creation and assignment of children because only available prefixes can be used to alter the skew and it is therefore necessary to do this only after the children have been assigned and each node's available prefix count therefore reflects this.

The generation then proceeds to the next phase.

4.2 Prefix ending

When this phase starts, child nodes have been created, assigned to current level's nodes and given a minimum amount of prefixes. At its end, all nodes from the current level have been assigned a number of prefixes according to the prefix length distribution.

This is the simplest phase. It starts by ending all prefixes in all childless nodes, as they have nowhere else to pass their prefixes to. After that it just randomly ends the number of prefixes required by the prefix length distribution and then the generation moves on to the next phase. This phase also attempts to minimize prefix nesting by semi-randomly picking nodes based on their prefix nesting and attempting to see if the prefix length distribution could still be satisfied should the picked nodes simply pass all of their prefixes to their children. The probability a node will be picked increases with its prefix nesting. This does not include childless nodes as all their prefixes must end regardless of other circumstances, but this does not really have much impact on prefix nesting as the prefix nesting of childless nodes is final and cannot increase further.

4.3 Distribution of prefixes to children

When this phase starts, all the nodes from the current level have had children as well as prefixes assigned to them. When it ends, all remaining available prefixes have been distributed to child nodes and the processing can move to the next trie level.

This phase starts by first passing all remaining available prefixes in one-child nodes to the node's only child, as that is the only possible remaining action for these nodes. After that, it only concerns itself with two-children nodes, as the distribution it attempts to satisfy is the skew distribution, which is only defined for two-children nodes.

Seeing as skew is defined as in Equation 3.1, calculating the effect of adjusting the ratio of light subtree weight and heavy subtree weight would be somewhat convoluted and counter-intuitive. The generator therefore uses an inverse skew sum as a target, which is defined as follows:

$$target\ sum = (1 - target\ skew) * two - children\ node\ count \tag{4.4}$$

This sum can be used as the target and the calculation and is equivalent to using the skew from the skew distribution as a target due to the following relationships:

$$skew = (\sum_{i=1}^{len(fractions)} (1 - fractions[i]))/len(fractions) =$$

$$= (len(fractions) - (\sum_{i=1}^{len(fractions)} fractions[i]))/len(fractions) =$$

$$= 1 - (\sum_{i=1}^{len(fractions)} fractions[i])/len(fractions)$$

and so

$$1 - skew = 1 - (1 - ((\sum_{i=1}^{len(fractions)} fractions[i])/len(fractions))) =$$

$$= (\sum_{i=1}^{len(fractions)} fractions[i])/len(fractions)$$

and finally

$$(1 - skew) * len(fractions) = \sum_{i=1}^{len(fractions)} fractions[i]$$

where $fractions$ is an array containing a fraction $\frac{light\ weight}{heavy\ weight}$ for each two-children node.

The generator then iterates over all two-children nodes and randomly assigns a light subtree weight to each of them. After that, it checks if there is a constraint for the next level specifying that there will be nodes with no children. If there are not enough low-prefix nodes, it could mean that it will be impossible to uphold the prefix length distribution at the next level. The generator therefore tries to ensure that there are enough nodes with low enough prefix count that the prefix length distribution can be adhered to at the next level. After that, it tries to minimize the error by looking through all the two-children nodes, attempting to calculate how close it can get to the desired skew, executing the best found adjustment and then continuing until no adjustment can get closer. Lastly, for each two-children node, the generator randomly picks which subtree is going to be the heavier and distributes the prefixes according to previous decisions. The level is now completely generated and the generation can move on to the next level.

4.4 Constraints

Each level has its own constraint. The constraint specifies how many childless nodes there were for a level, how many prefixes end at that level, the number of parent nodes at that level and the ratio of two-children nodes to parent nodes. Constraints are mainly used in the child node distribution phase. For example, when a constraint specifies that there are 12 childless nodes but no prefixes are supposed to end at the level, it means that the number of nodes at that level was wrong before. During the child distribution phase, when the generator is selecting the multiplier to apply to the ratio of two-children nodes to parent nodes, it simply makes the multiplier as large as it can. In case there is a constraint on the next level, the generator examines it. If it finds that the number of childless nodes at the next level would be larger than the number of prefixes to end at that level, it attempts to generate the children in a way that would avoid this. Such a situation can result from two different causes:

1. The multiplier was too high, which is the simple case. It can be rectified by successively lowering the multiplier, calculating the resulting number of children, the number of two-children nodes and one-child nodes that the next level would have based on that number of nodes. If, with those numbers, the number of childless nodes would be low enough for the generator to be able to satisfy the prefix length distribution as well as the branching probability distribution, the generation can proceed.

2. The multiplier was correct, but there were not enough multiprefix nodes, leading the generator to assume a number of parent nodes that was too low, leading to too many childless nodes. To find out if this was the case, the generator examines the number of childless nodes, the number of parent nodes, and the ratio of two-children nodes to parent nodes specified in the constraint. From these numbers, the generator calculates the multiplier that was used in the attempt that resulted in the creation of the constraint, as well as the maximum multiplier that could have been used. In case less than the maximum was used, the generator assumes that this could be rectified by making sure there are enough multiprefix nodes for the next level to be able to use a higher multiplier and therefore have fewer childless nodes.

   An example of this is when the generator creates 50 children, the branching probability is 0.2 and 7 prefixes are supposed to end at the next level, but there are only 8 multiprefix nodes at the next level. In such a case, the next level would find a ratio of $\frac{1}{5}$ and while adhering to this ratio, the generator would conclude that the multiplier can be at most 8 for a total of 40 parent nodes. But this would mean that the remaining 10 nodes must remain childless. To avoid this outcome, the generator can set a condition that there must be at least 9 multiprefix children, which would mean that during generation of the next level the multiplier can be set to 9 and there will be only 5 childless nodes. As long as the total number of prefixes in the 5 children with the lowest prefix count is not greater than 7, the generation of the next level can succeed.

Another part of generation that takes the constraints into consideration is the phase distributing the remaining available prefixes to the child nodes. It attempts to make sure that there are actually enough low-prefix nodes in the next level so that it can have the necessary number of childless nodes while also ending only the required number of prefixes. This can be a problem when, for example, the next level is supposed to have 3 childlesss nodes and only 3 prefixes are supposed to end at that length. If there are only two single-prefix nodes, it would be impossible to have 3 childless nodes while ending only 3 nodes. While deciding on how to allocate prefixes to child nodes in two-children nodes, the generator therefore checks if there is a constraint for the next level. If there is, it first examines one-child nodes to see if some of them do not pass a number of prefixes to their children that is sufficiently low for them to be used as childless nodes in the next level. Afterwards, taking into account the found childless nodes, the generator forces some two-children nodes to pass only a limited number of prefixes to one of their children so that the constraints can be satisfied.

For example, if the constraint indicates that there will be 5 childless nodes and that 10 prefixes must end at the next level, it will first try to iterate over the one-child nodes. For the sake of this example, suppose that one of them passes 3 prefixes to its child. In that case, it remains to find 4 nodes with total prefix count not exceeding 7. To this end, the generator creates a set of the two-children nodes' potential children that is sorted by weight. It keeps track of the sum of the weights of the 4 potential children with the lowest weights. It then randomly increases the skews of the parents of the children with lowest weights, as that should lower their weights. If one of them is the heavier child, its weight will increase, but that should eventually push it out of the nodes with the lowest heights and replace it with a child that is the lighter child of its parent and whose weight can then be lowered further.

Chapter 5

Implementation

During implementation of the generator described in the previous chapter, numerous problems were encountered. The generator was tested on the sets of prefix set parameters supplied with the ClassBench tool (12 parameter files with a source and destination prefix set each, for a total of 24 sets of prefix set parameters). The generator was able to always finish generating for 9 of those sets, for 9 sets it only finished sometimes, and for 6 sets it never succeeded. In the cases the generator did not run to completion, it either crashed or entered an infinite loop. The overview of how well it performed for different parameter sets can be found in Appendix A. After investigating the causes of these errors, it was found that rather than programming errors, they were errors in the algorithm itself.

For example, when the phase distributing prefixes to child nodes is attempting to ensure that there are enough low-prefix nodes to satisfy the prefix length distribution at the next level, there is no mechanism in place should this not be possible. The generator also does nothing in case it is unable to keep the difference between reached skew and desired skew donw to a reasonable level. This can, however, be highly likely since it can force many (even all) the two-children nodes of a level to be highly skewed in order for the generator to be able to adhere to the prefix length distribution and the branching probability distribution during generation of the next level. The generator also attempts to ensure that there are enough multiprefix children so that the next level can have enough two-children nodes to adhere to the branching probability distribution while also keeping the number of childless nodes low enough to be able to adhere to the prefix length distribution. The mechanism it uses to do so, however, only concerns itself with the next level. For the generator to be able to create enough multiprefix nodes, the current level must also contain enough high-prefix two-children nodes. For example, if a level with 4 one-child nodes and 3 two-children nodes is supposed to have 10 multiprefix children, each of the one-child nodes must have at least 2 prefixes itself and each of the two-children nodes must have at least 4 prefixes in order to be able to pass on 2 prefixes to each of its children. This can propagate to the previous level, where it could require even higher prefix counts and propagate to the previous levels again in this manner.

It would of course be possible to fix these problems, but it would, at the very least, require a significant rewrite of most of the generator, introducing further complexity, only to possibly find other problems. Due to the fact that the time to finish this work is limited, I have decided to implement a completely different approach to prefix set generation, which would not so much solve these problems as avoid them completely. This approach has the practical advantage of being simpler than the one described in the previous chapter. Instead of first completely generating one level, then moving onto the next one, the generator first

generates the basic structure of the trie and then alters it to get the desired skew. However, there is no implementation of a mechanism that would prevent excessive prefix nesting. At the end of the chapter, I will outline a way to possibly lower the maximum prefix nesting.

Most of the complexity of the proposed way of generating the trie stemmed from constraints whose purpose was to enable the lower levels of the trie to satisfy the prefix length distribution and the branching probability distribution. Due to the fact that it was this complexity that eventually led to the implementation failing, the method used to replace it was designed so that it would avoid the need for those constraint in the first place. It therefore first makes sure that the prefix length distribution and the branching probability distribution are satisfied and then it swaps nodes and prefixes around to satisfy the skew distribution as well. It does this by first attempting to move nodes and prefixes to satisfy the skew distribution. If at least one level cannot get within a certain margin of error of the target skew, the generator attempts to redistribute prefixes at all levels so that there are more prefixes in the subtrees of the two-children nodes. This should improve the skews that can be reached, as increasing the number of prefixes in the children of a two-children node leads to an increase in the range of skews it can reach. The generator also attempts to distribute prefixes evenly among the subtrees of two-children nodes. While the overall skew is a simple arithmetic mean, the individual skews are quotients and the increase of their possible ranges is much slower in nodes with a high prefix count than in nodes with a low prefix count. There were therefore two properties that the metric for this distribution should have. The first is that it should increase with the increase in the sum of weights of the monitored nodes, the second is that it should be higher when the weights are more evenly distributed. Both of these are properties of the geometric mean, which was therefore used as the metric. After the redistribution is done, the generator loops back to attempting to adjust the skews.

The generator does this a certain number of times, and if it still cannot reach satisfactory skews, it restarts the whole generation from the beginning. The number of restarts is also limited, and once that limit is reached, the generator just gives up and uses the last reached result.

5.1 Generation of trie structure

First, the generator creates the trie itself, that is the nodes that make up the trie. In the beginning, it does not do anything with the prefix count. The generation proceeds recursively from the root to the leaves.

It starts by generating nodes, the count of which is received from the previous level. It then looks at the prefix length distribution to find out how may prefixes can end at this level. For the last level, it simply checks if the number of nodes is not greater than the number of prefixes to end. If it is, the generator goes back to the previous level to try with a different number of nodes. Otherwise it just first gives each node one prefix as all nodes are leaf nodes and must therefore have at least one prefix. Then it randomly distributes the remaining prefixes of that length to the generated nodes. At all other levels, it first calculates the ratio of the count of two-children nodes to the count of all parent nodes. The generator then selects the highest multiplier that is possible with the amount of nodes currently generated at the current level. The number of children that would result from selecting this multiplier is calculated and the generator then attempts to recursively create the next level with the number of nodes generated being the number of children that would be generated with the selected multiplier. If the generation fails, the

multiplier is lowered and the generation of the next level is attempted again. In case the multiplier falls so low that there would be too many childless nodes to satisfy the prefix length distribution, the generation of this level also fails and the previous level must try again. The failure propagates in this manner until reaching a level that can reduce its multiplier while satisfying the prefix length distribution. When the generation of the lower levels succeeds, the generator proceeds to add the necessities to the current level. First, it gives each childless node one prefix. After that, it randomly assigns the nodes generated in the next level as the children of nodes from this level. The generation of the structure of the trie is then done, but there may still be a number of prefixes that should end at this level, so the generator randomly distributes these prefixes to the generated nodes. With that, the generation of the trie's structure for this level is successfully over.

5.2 Skew adjustment

The skew adjustment proceeds level by level, from the root to the leaves. From a high-level perspective, the algorithm is almost the same as the one described in Chapter 4. It simply finds the adjustment that minimizes the difference between the desired skew and the current skew, performs it, and then repeats the process as long as such adjustments exist. The process of altering the node skews is, however, completely different.

To adjust the skew of a given node, the generator first creates two sets of nodes, one for each of the subtrees, and initializes each to contain the root of its respective subtree. It then attempts to see how many prefixes could be transferred between the two sets in the direction which would get the skew closer to the desired skew. Afterwards, it replaces the nodes in each set with their children and repeats the process. The process of determining the best possible way to transfer prefixes between the sets works by first attempting to see how many prefixes could be taken from the nodes in the set that prefixes should be transferred from. It then tries which possible transfer would get the node the closest to the desired skew. Another way it can transfer prefixes between the sets is finding a pair of nodes that could be swapped between the sets in which the weight difference is the closest to the desired adjustment of the weights of the subtrees. This means that the generator can freely move child nodes and prefixes (of course only within the same level) in all descendants of the node whose skew is being adjusted, but it cannot move them in/out of the subtree rooted at the node being adjusted. This is because moving prefixes within the same subtree does not influence skews in the subtree's ancestors, but moving nodes in/out of the subtree would. That would be undesirable as the skew on the higher levels has already been adjusted.

In case the overall skew is too far from the desired value, the generator notes the current geometric mean of the weights of the children of two-children nodes in the given level and records it.

5.3 Adjusting skewable prefixes

In case the desired skew cannot be reached, the generator attempts to adjust the number and distribution of prefixes in the children of two-children nodes. This process starts at the leaves and proceeds to the root. It attempts to make the geometric mean of the weights of the children of two-children nodes larger than the one that was recorded when the skew adjustment failed.

To increase the metric, the generator first splits the nodes of the level into a set of nodes with a one-child parent and a set of nodes whose parents also have another child. The generator first attempts to increase the sum of weights of the nodes with a sibling by swapping high-prefix nodes without a sibling for low-prefix nodes with a sibling. If the geometric mean is still lower than the desired one, the generator attempts to even out the differences between the weights of the nodes with siblings. It does so either by directly transferring prefixes or by swapping their children. The child swaps can however be only performed when both nodes have the same number of children, as swapping children between a one-child node and a two-children node would alter the metric for the next level, which has already been processed and should not be altered this way again.

Chapter 6

Generation results

This chapter presents the results obtained by running the generator implemented in this thesis. It compares the results obtained from ClassBench, which serve as the benchmark for the performance of the solution created in this work, with the results obtained by running the final implementation of the algorithm described in this thesis (labeled as probgen2). Where applicable, the results from the first proposed algorithm of generation, which was eventually abandoned (labeled as probgen), are also presented. Unless otherwise noted, the values presented below are calculated from 10 runs of the generators. The metric used for aggregation of the data from multiple runs and comparison between the different generators is the Root Mean Square Error, abbreviated as RMSE. It is calculated using Equation 6.1

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(reached_i - target)^2} \tag{6.1}$$

where n is the number of generated prefix sets, *reached* is the value computed for the generated set and *target* is the value from the parameter file.

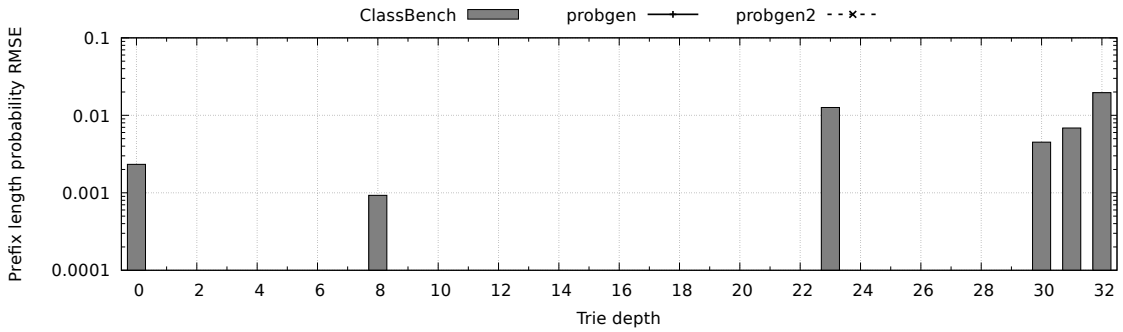

Figure 6.1: acl1 source prefix length distribution RMSE

RMSE is calculated for the characteristics specified in the parameter files, specifically the prefix length distribution, the branching probability distribution, and the skew distribution. Seeing as the charts show an error, in case the values match exactly, the error is equal to zero. The chart plotting the error then does not show the data lines for the new generators at all, as can be seen for example in Figure 6.1. The RMSE for both algorithms described in this thesis is not even shown in the chart, as the tool used for creating the charts from

the data (gnuplot) cannot show zero values in charts that use logarithmic scale. Not using logarithmic scale would solve this problem, but unfortunately, as will be shown below, the results obtained from the new generators are usually much better than those from ClassBench and the errors are smaller by orders of magnitude. Also, the errors sometimes vary by orders of magnitude even within the results from one generator. Using linear scale would therefore result in most values being extremely close to the bottom of the chart and the differences would be de facto invisible.

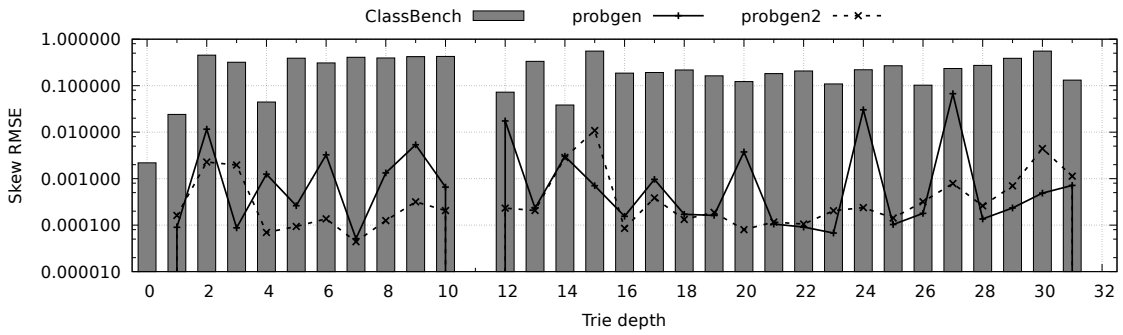

Figure 6.2: acl4 source skew RMSE

6.1 Prefix length

The prefix length distribution is always adhered to perfectly. Concerning the new generators, every chart plotting the prefix length RMSE therefore looks just like Figure 6.1. The prefix lengths will not be discussed further, as there is no interesting information.

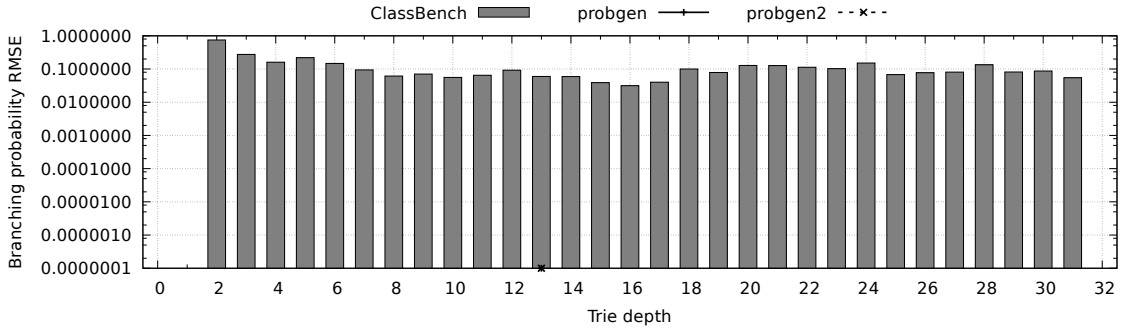

Figure 6.3: acl4 destination branching probability RMSE

6.2 Branching probability

The branching probability distribution is mostly adhered to perfectly as well, although the charts sometimes show a small error like in Figure 6.3. A look at the data files reveals that the error is exactly 1E-7, which is 1 unit in the last place in the precision used for

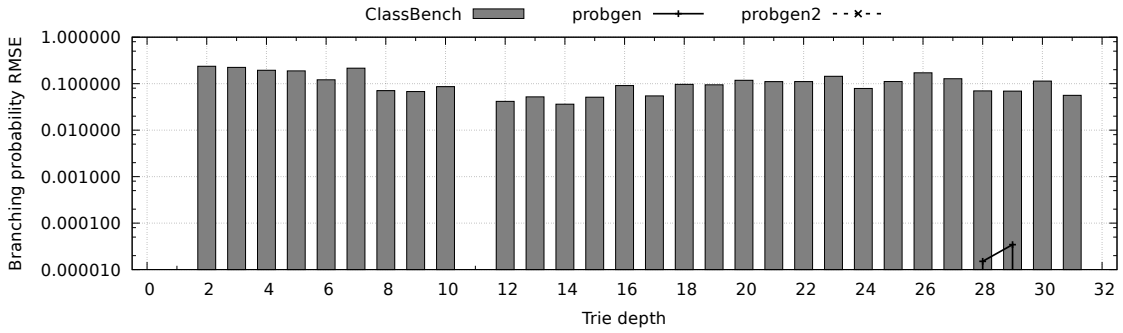

Figure 6.4: acl4 source branching probability RMSE

recording the prefix set statistics and therefore assumed to be rounding error rather than actual error.

There is, however, an exception. In Figure 6.4, there are errors at levels 28 and 29 that were not caused by rounding. These errors occured only during generation by probgen, described in section 4.1. Probgen2 was not affected. This is despite the fact that both of them use the same algorithm (described in Equation 4.1) to decide on the proportion of one-child nodes to two-children nodes. The main difference between the two generators in this regard is that the second one simply uses the number of nodes at the given level as the upper limit for the number of parent nodes, while the first one also uses the number of multiprefix nodes to possibly limit the denominator of the resulting ratio. If there were few enough multiprefix nodes to cause the generator to think that the possible number of parent nodes is lower than it should be, but not low enough to make adherence to the prefix length distribution impossible, the generation would actually continue. This is supported by the fact that the parameter file specifies that the branching probability should be 0.212871, which is closely approximated by the fraction $\frac{43}{202}$, but in two out of the nine runs that were successful, the branching probability was actually 0.212903, corresponding to the fraction $\frac{33}{155}$ (note the lower denominator). This results in a lower number of parent nodes, but the number of childless nodes in these runs was 58, which still made it possible to end only 70 prefixes as per the prefix length distribution. The affected runs were numbered 3 and 8.

This can happen because the generator merely looks for a ratio that is the closest to the desired brobability, it does not require equality, as doing that with floating point numbers tends to be quite dangerous. An example of rounding error was given earlier and the statistics in the parameter files are represented using a fixed format of 9 digits with the 1 to the left and 8 to the right of the decimal point. This is not enough to make IEEE-754 single-precision numbers roundtrip, as that requires printing 9 **significant** digits [5].

6.3 Skew

The skew distribution is not adhered to as well as the other distributions. Nevertheless, the results obtained by the new generators are still, with a few exceptions, consistently better than those acheived by ClassBench, for example Figure 6.5. Probably the worst results obtained by the new generator can be seen in Figure 6.6, particularly near the root, This is one of the sets for which probgen did not finish at all, which is why it is not presnt in the

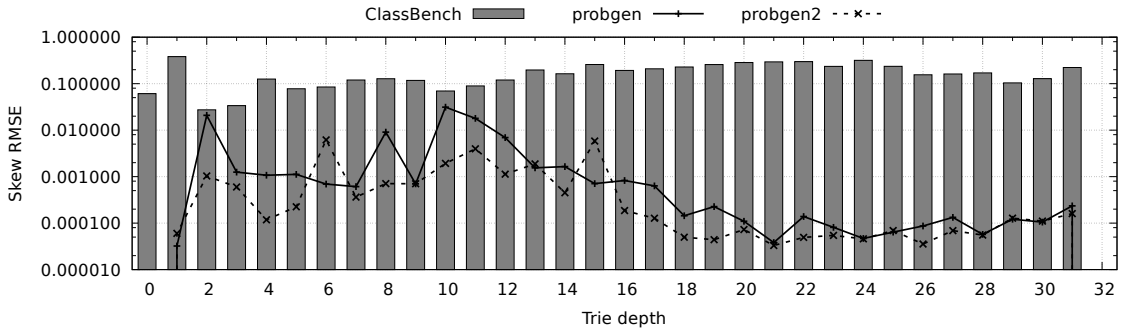

Figure 6.5: acl4 dest skew RMSE

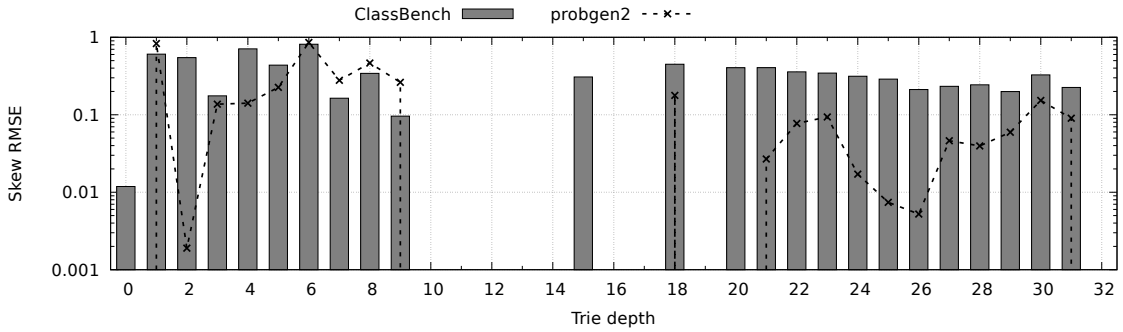

Figure 6.6: fw1 source skew RMSE

chart. Overall, the results are still better than those acheived by ClassBench, but at some depths (particularly 1 and 6 through 9) they are worse.

As for the comparison between the two new generators, as can be seen for example in Figure 6.7, they generally produce results of similar quality, and often even follow the same trends. This is expected, as from a high-level perspective, the ways in which the two generators attempt to reach the desired skew are quite similar (as has been noted in section 5.2).

There are some exceptions, for example Figure 6.8, in which the originally proposed generation algorithm has significantly better results, or Figure 6.9, in which the new algorithm performs better.

Now for the parameter sets which the first generator had trouble with. First, looking at the sets for which the generator failed only sometimes. In some, such as Figure 6.10, probgen2 had no trouble at all and generated results of similar quality to the ones acheived by probgen. However, there were several sets, for example Figure 6.8, in which the results of probgen2 were much worse than the ones acheived by probgen. This may, however, be explained by the fact that the times probgen did not finish were not included in the RMSE calculation, and the results may therefore look better as the runs when probgen suffered a complete failure were simply discarded.

Among the sets where probgen failed every time, probgen2 also produced bad results in three of the cases, for example Figure 6.6. In the other three, the results were quite good, such as Figure 6.11.

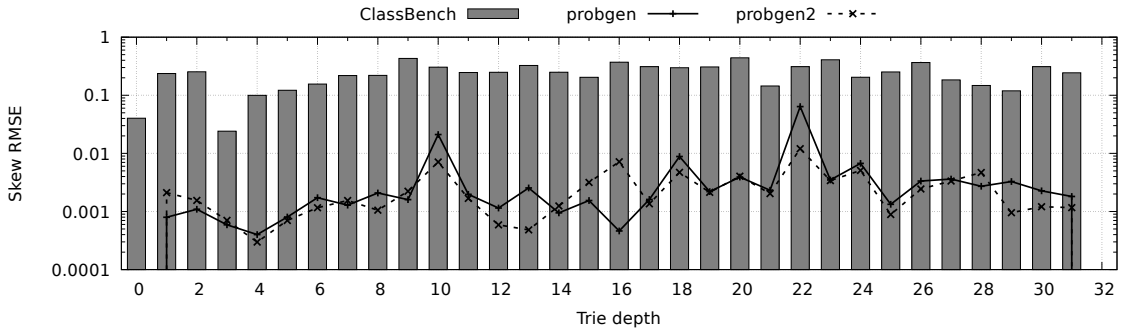

Figure 6.7: acl1 dest skew RMSE

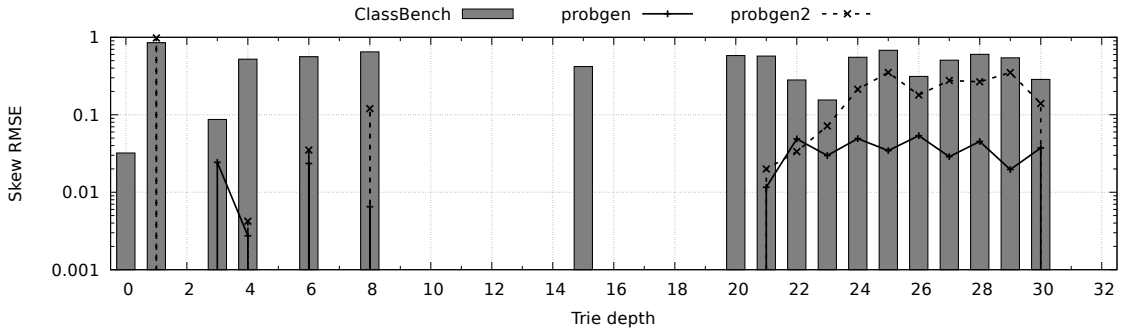

Figure 6.8: fw3 source skew RMSE

6.4 Prefix nesting

Probgen attempted to keep prefix nesting low, but it did not enforce it in any way. Probgen2 does not implement any way to limit prefix nesting. As a result, the new implementation often creates prefix sets with a prefix nesting that is much higher than was specified. The error can be as high as 9 or more than double the desired prefix nesting. In probgen, which attempts to keep it lower, the error tends to be approximately half the value it is in probgen2.

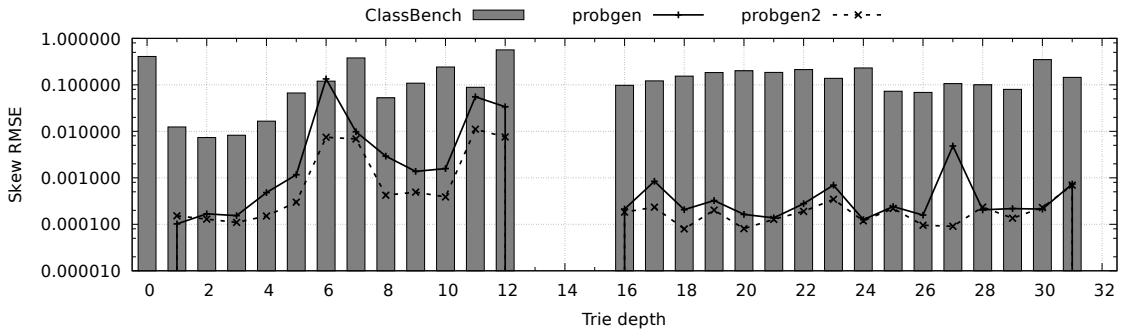

Figure 6.9: acl3 dest skew RMSE

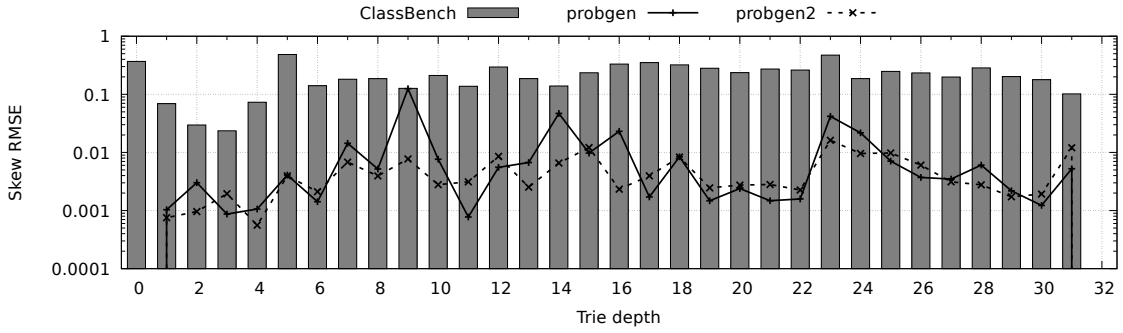

Figure 6.10: acl2 dest skew RMSE

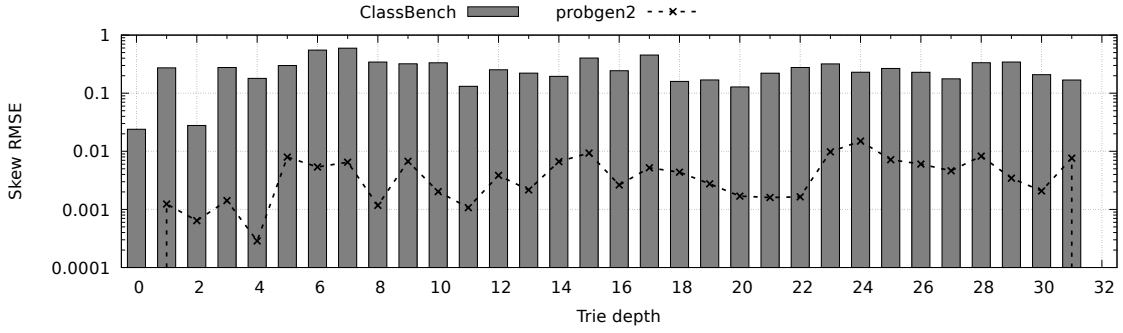

Figure 6.11: acl2 source skew RMSE

Chapter 7

Conclusions

The goal of this thesis was to propose and implement an algorithm for generating prefix sets that would reach better results than ClassBench.

The generators implemented in this thesis reach perfect results for prefix length. For branching probability, probgen almost always reaches perfect results and in the rare cases when it does not, it deviates from them only by a small amount. Probgen2 always reaches perfect results for branching probability. For the skew distribution, both generators presented in this thesis do not reach perfect results, but the error still tends to be orders of magnitude lower than the error in ClassBench's results. For prefix nesting, probgen2 deviates from the target by a rather large margin while probgen also generates prefix sets with larger prefix nesting, but not by as much. ClassBench never crosses the prefix nesting threshold.

Overall, ClassBench fulfills one parameter perfectly while often significantly deviating from the other parameters. The generators in this thesis fulfill two parameters perfectly or nearly perfectly, one parameter better than ClassBench by orders of magnitude and one parameter significantly worse.

7.1 Future work

It should be possible to add a way to limit prefix nesting into the new generator. After the generation finishes, the trie would be traversed and the paths with a high number of prefix nodes would be identified. The generator would then attempt to transfer the prefixes from the paths with a high amount of prefix nodes to the paths with fewer prefix nodes. Of course, this would have the potential to change the skews, so this would have to be mitigated. Possible ways include the following:

1. If prefixes were transfered between nodes, the generator would then find a descendants/ancestors of the nodes which also contain prefixes, are at the same level, and can be reached from the original node without passing through a two-children node. It would then transfer the same amount of nodes between these descendants/ancestors. This would preserve the skews reached during previous generation.

2. The generator could just check how much the skew distribution would be affected and only move the prefixes if it would not lead to skew error becoming too large.

Another possibility would be to combine the two algorithms, taking the best from each. The result would generate the trie structure much like the second algorithm does, but it

would only add prefixes where strictly necessary, i.e., would give exactly one prefix to each childless node. The parent-child relations would be created from the leaves to the root and would be made such that the skew would be as close as possible to the desired skew. Afterwards, the trie would be traversed from the root to the leaves, with prefixes being assigned to nodes and distributed to children in much the same way as was used in the original proposal. This would also make it possible to use the same mechanism that was used in the first generator to keep the prefix nesting within reasonable values.

Bibliography

[1] RIS Raw Data. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data. accessed: 2017-02-12.

[2] The standard trie. http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/trie01.html. accessed: 2017-05-12.

[3] University of Oregon Route Views Project. www.routeviews.org. accessed: 2017-05-12.

[4] Chendi, S.: [Apnic-announce] APNIC IPv4 Address Pool Reaches Final /8. 2011. accessed: 2017-02-12.
Retrieved from: http://mailman.apnic.net/mailing-lists/apnic-announce/archive/2011/04/msg00002.html

[5] Dawson, B.: Float Precision–From Zero to 100+ Digits. 2012. accessed: 2017-05-07.
Retrieved from: https://randomascii.wordpress.com/2012/03/08/float-precisionfrom-zero-to-100-digits-2/

[6] Knuth, D. E.; Graham, R. L.; Patashnik, O.: *Concrete mathematics : a foundation for computer science*. Reading, Mass: Addison-Wesley. 1994. ISBN 0-201-55802-5.

[7] Ruiz-Sanchez, M.; Biersack, E.; Dabbous, W.: Survey and taxonomy of IP address lookup algorithms. *IEEE Network*. vol. 15. March 2001: pp. 8 – 23.

[8] Taylor, D. E.; Turner, J. S.: ClassBench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking*. vol. 15. June 2007: pp. 499 – 511.

Appendices

Appendix A

List of prefix set parameter sets by success of first version of generator

Note that as the generator uses randomly-seeded PRNG, your results may vary. These are results from 10 runs on each set

- Successful generation:
  1. acl1 source
  2. acl1 dest
  3. acl3 source
  4. acl3 dest
  5. acl4 dest
  6. fw3 dest
  7. ipc1 source
  8. ipc1 dest
  9. ipc2 source
- Sometimes uccessful generation:
  1. acl2 dest (4 successful)
  2. acl4 source (9 successful)
  3. acl5 dest (6 successful)
  4. acl5 source (9 successful)
  5. fw1 dest (6 successful)
  6. fw3 source (5 successful)
  7. fw5 dest (9 successful)
  8. fw5 source (5 successful)
  9. ipc2 dest (6 successful)
- Unsuccessful generation:
  1. acl2 source

2. fw1 source
3. fw2 dest
4. fw2 source
5. fw4 dest
6. fw4 source